Bolt Beranek and Newman Inc.

Report No. 4816

Development of a Voice Funnel System

Quarterly Technical Report No. 10
1 November 1980 to 31 January 1981

March 1982

Prepared for:
Defense Advanced Research Projects Agency

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A112 485	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Development of a Voice Funnel System Quarterly Technical Report No. 10		5. TYPE OF REPORT & PERIOD COVERED Quarterly Technical 1 Nov. 80 - 31 Jan 81
		6. PERFORMING ORG. REPORT NUMBER 4816
7. AUTHOR(s) R. D. Rettberg		8. CONTRACT OR GRANT NUMBER(s) MDA903-78-C-0356
9. PERFORMING ORGANIZATION NAME AND ADDRESS Bolt Beranek and Newman Inc. 10 Moulton Street Cambridge, MA 02238		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order No. 3653
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209		12. REPORT DATE March 1982
		13. NUMBER OF PAGES 78
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Voice Funnel, Digitized Speech, Packet Switching, Butterfly Switch, Multiprocessor

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This ~~Quarterly~~ Technical Report covers work performed during the period noted on the development of a high-speed interface, called a Voice Funnel, between digitized speech streams and a packet-switching communications network.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Report No. 4816 Bolt Beranek and Newman Inc.

DEVELOPMENT OF A VOICE FUNNEL SYSTEM

QUARTERLY TECHNICAL REPORT NO. 10
1 November 1980 to 31 January 1981

March 1982

This research was sponsored by the
Defense Advanced Research Projects
Agency under ARPA Order No.: 3653
Contract No.: MDA903-78-C-0356
Monitored by DARPA/IPTO
Effective date of contract: 1 September 1978
Contract expiration date: 31 December 1981
Principal investigator: R. D. Rettberg

Prepared for:

Dr. Robert E. Kahn, Director
Defense Advanced Research Projects Agency
Information Processing Techniques Office
1400 Wilson Boulevard
Arlington, VA 22209

Table of Contents

1. Introduction..... 1
2. BIO Module Architecture and Facilities..... 3
3. Asynchronous Channels..... 8
3.1 Asynchronous Receiver Operation..... 8
3.2 Asynchronous Transmitter Operation..... 11
3.3 Modem Interface Signals..... 12
3.4 Asychronous Channel Control Registers..... 13
3.4.1 Status Register..... 14
3.4.2 Transmitter Holding Register..... 16
3.4.3 Receiver Holding Register..... 17
3.4.4 Input FIFO Oldest Data Entry..... 17
3.4.5 Mode Control Register A..... 18
3.4.6 Mode Control Register B..... 19
3.4.7 Command Register..... 20
3.4.8 Interrupt Enable Mask Registers..... 21
3.4.9 Device/Version Number..... 22
3.4.10 ID/Interrupt Vector Registers..... 22
3.4.11 DMA Address Register..... 23
3.4.12 Character Block Counter..... 24
3.4.13 Conditions for Normal Operation..... 24
4. Synchronous Channels..... 26
4.1 Interface Protocols..... 27
4.2 Bit-Oriented Protocol..... 28
4.3 Channel Control Blocks..... 30
4.4 Device Control Blocks..... 34
4.5 Enabling the Channel..... 40
4.6 Looping the Channel..... 41
4.7 Fatal Errors..... 41
4.8 The Reset Function..... 43
4.9 Real Time Clock..... 45
4.10 Transmitter..... 46
4.10.1 Channel Control Block Flags..... 48
4.10.2 Packet Transmit Times..... 49
4.10.3 Maintaining the Transmit Queue..... 51
4.10.4 Initialization..... 52
4.11 Receiver..... 53
4.11.1 Backup Buffers..... 57
4.11.2 Counting Free Buffers..... 58
4.11.3 Signalling Packet Arrival..... 60
4.11.4 Maintaining the Receiver Queue..... 62
4.11.5 Initialization..... 64
4.12 Byte Ordering..... 64
5. Appendix: Algorithms Used in the BIO..... 66

FIGURES

BIO Architecture..... 4
Device Control Block Physical Address..... 7
Asynchronous Control Registers..... 14
Bit-Oriented Frame Format..... 28
Channel Control Block Structure Definition..... 32
Synchronous I/O Device Control Block..... 35
Output of Three Buffers Into Two HDLC Frames 48
Input of Three HDLC frames into five buffers 55

TABLES

BIO Features..... 5
Channel Control Block Status Flags..... 32
Channel Control Block Fields..... 33
Device Control Block Common Fields..... 36
DCB Transmitter and Debugging Fields..... 37
Device Control Block Receiver Fields..... 38
DCB Receiver Control Fields (continued)..... 39
Valid Settings for the Field cs_enable..... 40
Synchronous Channel Error Codes..... 42
Procedure for Resetting a Synchronous Channel..... 44

1. Introduction

This Quarterly Technical Report, Number 10, describes aspects of our work performed under Contract No. MDA903-78-C-0356 during the period from 1 November 1980 to 31 January 1981. This is the tenth in a series of Quarterly Technical Reports on the design of a packet speech concentrator, the Voice Funnel.

The hardware design of the Butterfly I/O module (the BIO) was described previously in Quarterly Technical Report Number 7. The BIO contains a microcoded bit-slice processor quite similar to the PNC. Working in conjunction with the PNC, each BIO module provides four low-speed asynchronous telecommunications channels and four high-speed synchronous channels for use in the Butterfly Multiprocessor. This report presents the architecture of the BIO module, and defines the functions which control the asynchronous and synchronous channels.

The BIO has been designed specifically to meet the needs of the Voice Funnel application. It is expected that other I/O modules will be designed as required to meet other needs in further projects. The BIO is a compromise between the need in the Voice Funnel for a number of high-speed synchronous interfaces (to the PSAT and LEXNET), and for a number of standard asynchronous interfaces (for system control and direct vocoder connection).

We should note that we use the notational conventions of the "C" programming language for distinguishing between decimal and hexadecimal representations for numerical quantities. Thus we will refer to the quantity "sixteen" as either "16" or "0x10". In addition, we are numbering bits with bit 0 on the right or least significant position in accordance with the MC68000 documentation.

2. BIO Module Architecture and Facilities

The BIO module has been designed to offload most of the I/O processing, control, and high speed data transfer from the Butterfly Processor Node. As shown in Figure 1, the BIO consists of an interface to the Butterfly I/O bus (the BIOLINK), an 8 mhz 16-bit 2901 microprocessor with 1024 words of scratchpad memory, four Signetics 2661-1 Enhanced Programmable Communications Interfaces, four Signetics 2652 Multi-Protocol Communications Controllers, and various support circuitry. The BIOLINK is capable of supporting up to four BIO modules.

The BIO microprocessor responds to PNC commands, keeps pointers and state information for each device in its scratchpad memory, controls all data transfers, provides scratchpad FIFO buffering for each device, requests DMA memory accesses from the PNC, and uses the PNC to interrupt or post Events to the MC68000. The BIO microcode provides many unusually sophisticated features to the application program. These features have been chosen to facilitate the Voice Funnel application, but are also expected to be suitable for other telecommunication applications using line protocols supported by the 2661 and 2652 devices. The principal features of the BIO module are summarized in Table 1.

Every I/O channel is associated with a group of registers in Subspace One. These registers hold flags and parameter values that control the behavior of the I/O microcode and cause it to

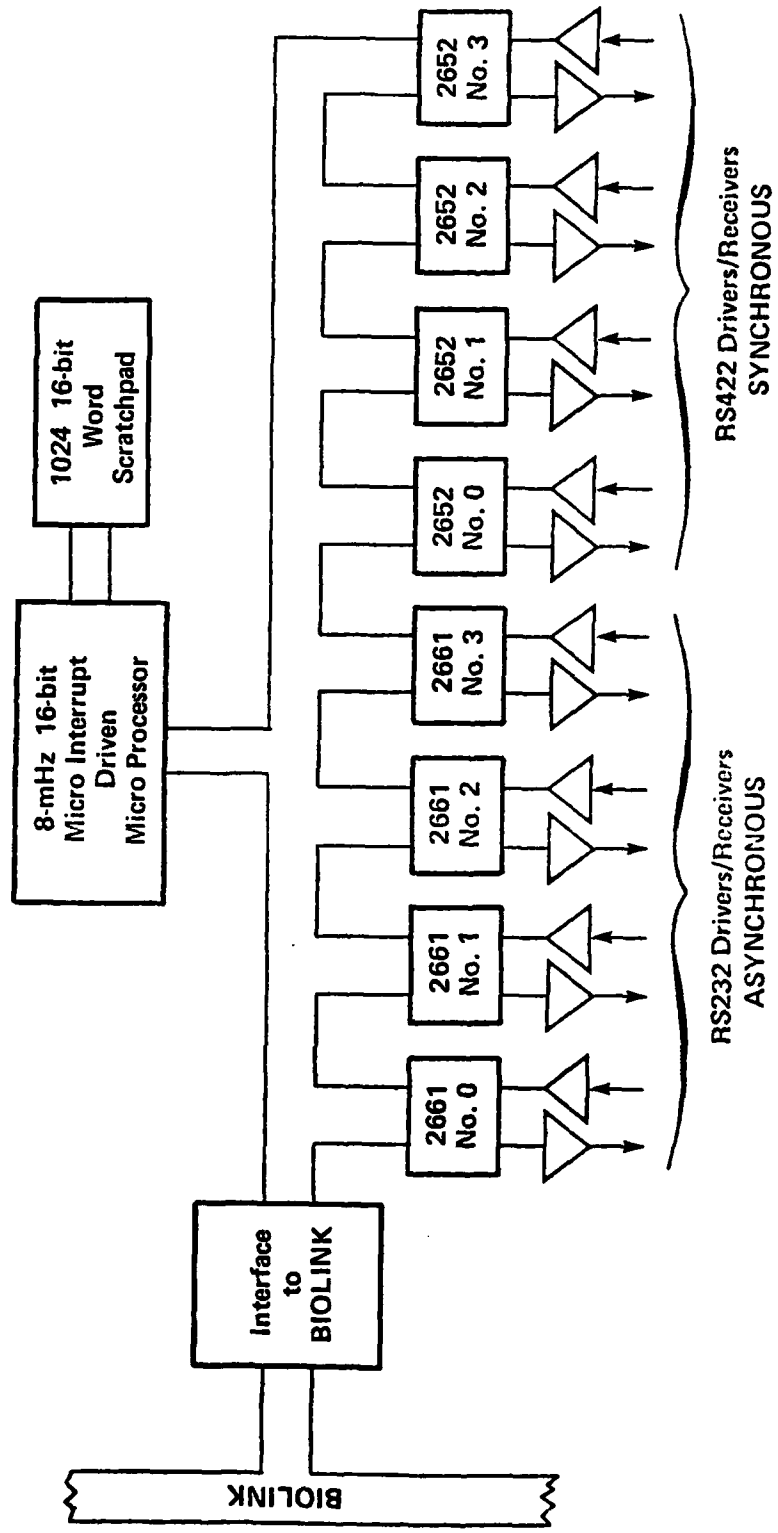

Figure 1 - BIO Architecture

Asynchronous Channel Features

- Data rates up to 19.2 kbps
- Interrupt vectors for up to three separate events
- Program controlled hardware echo of received characters
- 32-character FIFO for input characters
- DMA controller for output using either native mode or PDP-11 style byte order
- Break detection and generation
- Complete set of line conditioning signals available for modem control
- EIA RS-232-C compatible inputs and outputs
- Local or remote maintenance loop back mode

Synchronous Channel Features

- Maximum data rate of 2 Mbps
- Automatic detection and generation of special bit-oriented protocol FLAG, ABORT, and GA control sequences
- 64 character input and output FIFOs to improve latency
- DMA controller for both input and output
- Chained DMA control blocks to provide fast buffer swaps without Processor Node intervention
- Guaranteed minimum time between transmitted messages
- Direct posting of operating system Events
- EIA RS-422 compatible inputs and outputs

Table 1. BIO Features

initiate certain operations. We will refer to this group of registers as the Device Control Block (DCB) of the channel. The layout of the physical address of a DCB register is given in Figure 2. With the exception of the subspace field, which is always one, the high order eighteen bits of the DCB physical address are currently ignored by the Processor Node hardware. As with all entities that are addressed through Subspace One, a Processor Node can only access Device Control Blocks that are local to it. The high order bit of the "channel" field of the DCB base address distinguishes between synchronous and asynchronous channels. For synchronous channels, it is a zero. For asynchronous channels, it is a one. The "protection" bit is used only for asynchronous Device Control Blocks. It is used to distinguish between fields that are accessible in kernel mode only (protection bit = 1) and those that are accessible in both kernel and user mode.

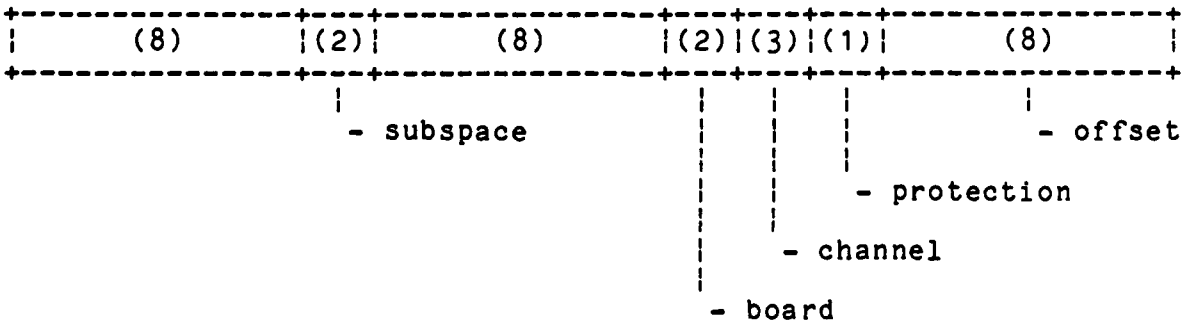

Figure 2 . Device Control Block Physical Address

The PNC generates a hardware reset signal when power comes up, when the MC68000 halts, when the PNC receives a reset message from a remote Processor Node, and when the hardware reset switch on the Butterfly clock card is cycled. In addition to resetting the processor node hardware, this signal resets the hardware on each module connected to the BIOLINK. This signal is the only mechanism for resetting the asynchronous channels on the BIO. Each synchronous channel also has a facility for resetting that channel individually.

3. Asynchronous Channels

The electrical signal level protocol supported by the BIO asynchronous channels conforms to EIA standards RS-232-C and RS-423. At the link protocol level, all the asynchronous channel services are provided by the 2661 devices; the 2661 also provides a synchronous mode, which we do not plan to use in the Voice Funnel application. We review the asynchronous-mode operation of this device to bring all the relevant information together in a single report. For further information see the Signetics 2661-1 documentation. We also describe the associated features and control facilities provided by the BIO microcode.

3.1 Asynchronous Receiver Operation

On each of the four asynchronous input channels, incoming signals are level-converted from RS-232-C by 26LS32 receivers, and fed to Signetics 2661-1 Enhanced Programmable Communications Interface devices (EPCIs). These 28-pin MOS/LSI circuits perform all the functions necessary for double buffered asynchronous character assembly (and also transmission; see next section).

The receiver section of the EPCI samples the line at 16 times the nominal bit rate of the signal expected on that line. Upon detection of a Mark-to-Space transition, the EPCI waits 8 clock pulses, then checks the line. This would be at the center of a normal start bit. If the line has returned to Mark, the

receiver assumes that there was noise on the line, and returns to its idling state, ready to detect another Mark-to-Space transition. If the line is at Space, the receiver assembles a data character by sampling the line at 16 clock tick intervals from the center of the start bit. The number of bits assembled is determined by the character length information entered into a 2661 mode register. If parity checking has been enabled, the receiver computes the parity of the character just received and compares it with the parity bit received. One stop bit is required, or a framing error is reported. Continuous Space is detected as an all-0's character with framing error; only one character is transferred as long as the condition persists.

Immediately following the receipt of the last bit, the data is transferred into the Receive Data Holding Register, and three status register bits are updated. The Receiver Ready (RxRDY) bit is set to indicate that the receiver holding register is not empty. If RxRDY was set at the time of the transfer, the Overrun Error (OE) bit is set, thus indicating that the previous character loaded into the receive holding register was not read by the microprocessor. The Parity Error bit (PE) is set when the parity sense differs from the parity bit.

At least once every 250 microseconds, the BIO microprocessor examines the status register of each EPCI. If RxRDY is set, it loads both the status and the receive holding register data into a 32-word first-in, first-out (FIFO) buffer and clears the RxRDY

bit. It further sets a FIFO Not Empty (FNE) bit in the EPCI controller status register.

A level four interrupt request is issued to the MC68000 upon loading the FIFO if the FIFO-Not-Empty Interrupt enable bit is set. The BIO microprocessor takes care of serializing the enabled FIFO-not-empty events by allowing only one interrupt to be pending at a time and scanning from the last interrupting channel.

The character length, parity sense, number of stop bits, etc., that will be used by the EPCI to perform the above operations are stored within each EPCI in a Mode Control register. This 16-bit register is addressable on a read/write basis by the MC68000.

Received characters may contain up to 43% distortion on any one bit due to the sampling rate. However, the overall bit rate must be accurate. Errors in bit rate are cumulative such that when the receiver samples the first stop bit to see if it is a Mark, the error accumulated by that time must not exceed 43% of a bit time. Assuming the reception of eight data bits plus parity, 4.2% speed distortion would be permissible.

Speed distortion of any amount poses severe problems in the auto-echo mode. However, if a terminal sends to the BIO at a slightly slower rate than the BIO retransmits, the auto-echo problem is solved. In computing speeds, one may assume the BIO

receiver clock to be accurate within 0.05% except at 1800 and 2000 baud where the error is -0.24% and -0.31% respectively.

3.2 Asynchronous Transmitter Operation

For each of the four asynchronous output channels, signals are generated by the output side of a Signetics 2661-1 EPCI, inverted, and then converted to RS-232-C by 26LS30 transmitter chips. The EPCIs perform all the functions necessary for double buffered asynchronous character transmission. The transmitter section of the EPCI holds the serial output line at a Marking state when idle. When a character is to be transmitted, the EPCI will generate a start bit (Space) within one-sixteenth of a bit time. The start bit and all subsequent data bits are a full bit time each. The start Space is followed by 5, 6, 7, or 8 data bits, as determined by control bits in the Mode Control register. The data bits are presented to the output least-significant bit first. The parity bit, if parity generation is enabled, is calculated by the transmitter and included after the last data bit, but before the stop sequence (one or more Mark bits). It is possible to send a break (continuous Space) by setting Command Register bit 3.

There are two mechanisms by which the MC68000 can cause a sequence of characters to be transmitted. Using the first, the MC68000 would load each character into the transmitter holding

register after determining (either by interrogating a status bit or as a result of an interrupt resulting from the status bit assertion) that the transmitter holding register is empty. Using the second, the MC68000 would set up a block in its local memory containing the sequence of characters, and give the BIO microprocessor its starting location and size. At least once each 250 microseconds the BIO microprocessor examines the status register of each EPCI. If the Transmitter Hold Register Ready (TxRDY) bit is set, it accesses the next byte from memory, loads it into the transmitter holding register, and updates the address pointer and byte counter. When the byte counter reaches zero, an interrupt request on level four is issued to the MC68000 if an interrupt enable bit is set. The BIO microprocessor takes care of serializing various events which may cause interrupts. The order of bytes in a word can be selected by a bit in Mode Control Register B.

The transmitter timing is derived by the same clock division logic as the receiver and is therefore accurate to 0.05% except at the 1800 and 2000 baud rates.

3.3 Modem Interface Signals

In addition to its receiver and transmitter sections, each EPCI provides two input and two output signals for controlling modems. All signals (including receiver input and transmitter

output) conform to the EIA RS-232-C and CCITT electrical specifications.

The modem interface signals provided are: Data Carrier Detect (DCD), Data Set Ready (DSR), Request to Send (RTS), and Data Terminal Ready (DTR). A level four interrupt may be enabled if any change occurs on the DCD or DSR lines from the modem.

3.4 Asychronous Channel Control Registers

The 17 control registers associated with each of the four asynchronous channels are located at fixed addresses as described in Section 2 and shown in Figure 3.

Offset	Content	Initially
000	Status Register	0x0100
002	[Tx Holding Register]	
004	[Rx Holding Register]	
006	[Input FIFO Oldest Entry]	
008	Mode Control Register A	0xFEBA
00A	Mode Control Register B	0
00C	Command Register	0
00E	Interrupt Enable Mask A	0
010	Interrupt Enable Mask B	0
012	Interrupt Enable Mask C	0
01E	Device and Version Number	00vv ddv.v
100	Id/Vector For Mask A	0
102	Id/Vector For Mask B	0
104	Id/Vector For Mask C	0
106	High 8 Bits Of Block Address	0
108	Low 16 Bits Of Block Address	0
10A	Character Block Byte Counter	0

Figure 3 . Asynchronous Control Registers

Each of these registers is described in detail in a section below.

The term "offset" refers to the difference between the address of the first register on a device block (i.e. the Status Register) and the register in question. Thus the Character Block Byte Counter register is located at offset 0x10A.

Individual I/O devices may be mapped into application program address spaces, to allow them to be used with minimal operating system intervention. The upper group of registers (above offset 0x100) contains privileged control information, which must be set up by the operating system if system integrity is to be maintained. All registers are read/write unless otherwise specified. Writing a read-only bit has no effect on the bit. Reading a write-only bit returns a zero. All interrupt requests are at level 4. Thus, the MC68000's interrupt priority level must be three or less to enable an interrupt from the BIO modules.

3.4.1 Status Register

This read-only register is divided into two bytes. The low order byte is read directly from the 2661 status register; the high order byte contains microprogram state information. Each of the three Interrupt Mask Registers defined below allows the application program to specify a set of bits in the status

register; if any of these bits are set, the corresponding interrupt will be requested by the interrupt polling microcode if the interrupt polling microcode has been enabled.

Bit 0: Transmitter holding register empty (TxRDY)

TxRDY is clear when the transmit holding register has been loaded and the data has not been transferred to the transmit shift register. TxRDY is set when the transmit holding register is ready to accept data. It is set initially when the transmitter is enabled. It is not set when the automatic echo or remote loop back modes are programmed.

Bit 1: Receiver holding register has data (RxRDY)

RxRDY is set when a character has been assembled and loaded into the receiver holding register. RxRDY is cleared when the MC68000 or BIO microprocessor reads the receiver holding register or when the receiver is disabled.

Bit 2: Change in DSR/DCD or transmit shift register empty

This bit is set when the transmit shift register has completed transmission of a character and no new character has been loaded into the transmit holding register, or when there has been a change of state in either the DSR or DCD inputs (and either the receiver or transmitter is enabled). This bit is cleared when the status register is read. If the status register is read twice and the bit is set while DSR and DCD remain unchanged then both the transmit holding register and shift register are empty. It can be used either to wait for a character to be completely transmitted, or, by disabling the transmitter while enabling the receiver, to wait for a change in modem state.

Bit 3: Parity Error (PE)

PE when set indicates a receiver parity error. PE is cleared when the receiver is disabled and by the reset error command.

Bit 4: Overrun Error (OE)

OE when set indicates that a previous character

loaded into the receiver holding register was not read before it was overwritten by a new character. OE is cleared when the receiver is disabled and by the reset error command.

Bit 5: Framing Error (FE)

FE when set indicates that the received character was not framed by a stop bit. FE is cleared when the receiver is disabled and by the reset error command.

Bit 6: Data Carrier Detect (DCD)

DCD is the Data Carrier Detect signal as sourced by an attached modem.

Bit 7: Data Set Ready (DSR)

DSR is the Data Set Ready signal as sourced by an attached modem.

Bit 8: Transmitter DMA Done (TDONE)

TDONE is set by microcode when the Character Block Byte Counter (CBBC) register is zero. TDONE is cleared when the CBBC is loaded with a non-zero value. When set, TDONE indicates that another block of characters may be transmitted via the DMA.

Bit 9: Input FIFO Not Empty (IFNE)

IFNE is set when input FIFO loading is enabled and a word is loaded into the input FIFO. IFNE is clear when no more words remain in the input FIFO.

Bits 10-15: Not used.

3.4.2 Transmitter Holding Register

Setting this write-only register loads a data character into the EPCI's transmitter holding register (from bits 7-0). The bits of the character are presented to the serial line low-order bit (bit 0) first. Bits 15-8 are not used. Status Register bit SR0 must be set in order for this operation to succeed.

This register should not be used if a DMA transfer is in progress to this device.

3.4.3 Receiver Holding Register

Bits 7-0 of this read-only register come directly from the EPCI; they are the last complete character assembled by the EPCI. If the character length specified is less than 8 bits, the character will appear right justified and the unused high order bits will be zero. The first data bit received will be loaded into RHR0, the second into RHR1 and so on. Bits 15-8 are always zero.

This register is useful if the microcode receiver polling feature has been disabled; see Mode Control Register B. Otherwise reading this register will usually return a copy of the last character placed into the FIFO, but occasionally may "steal" a character from the 2661 before the microcode can place it in the FIFO. In general, this register should not be used when the receiver polling feature is enabled.

3.4.4 Input FIFO Oldest Data Entry

This 16-bit read-only register has the oldest entry in the 32-word input FIFO, or zero if the FIFO is empty. When the MC68000 reads this register, the character is removed from the

input FIFO. If after removal, no more entries remain in the FIFO, the Input FIFO Not Empty bit is cleared. In order for characters to appear in the input FIFO, the microcode receiver polling feature must be enabled; see Mode Control Register B.

Bits 7-0: The character read from the EPCI's receiver holding register when the BIO microprocessor detected that the receiver holding register was not empty.

Bits 15-8: Bits read from bits 7-0 of the Status Register at the time when the BIO microprocessor detected that the receiver holding register was not empty. These status bits allow the application program to detect exactly at what point in a character stream unusual events occurred.

3.4.5 Mode Control Register A

Mode Control Register A is used to set the EPCI's two 8-bit Mode Registers. Reading this register returns a copy of the information last written into the EPCI. See the Signetics literature for further information.

Bits 1-0: Mode and baud rate factor (must be set to 10)

Bits 3-2: Character length (not including parity bit)

00 = 5 bits	10 = 7 bits
01 = 6 bits	11 = 8 bits

Bit 4: Check parity and report parity errors on input; add a parity bit on output.

Bit 5: Even parity (if parity is enabled)

Bits 7-6: Stop bit length

01 = 1 stop bit
10 = 1.5 stop bits
11 = 2 stop bits

Bits 11-8: Select baud rate settings for the 2661-1 (the 2661-2 and -3 differ, and are not used in the BIO)

0000 =	50 baud	1000 =	1050 baud
	75		1200
	110		1800
	134.5		2000
0100 =	150	1100 =	2400
	200		4800
	300		9600
	600		19200

Bits 15-12: Must be xx11; 1111 is recommended

3.4.6 Mode Control Register B

Mode Control Register B is used to control how the channel is serviced by the BIO microprocessor. These bits do not affect the EPCI directly. It is possible to use a channel on a character-at-a-time basis without enabling data polling; it is also possible to use a channel by testing its status periodically from the application program, without enabling interrupt polling in the BIO microprocessor.

Bit 0: Select PDP-11 style transmitter DMA addressing

If this bit is set, the transmitter sends the right-hand (odd address) byte of each data word before sending the left-hand byte. This mode is useful if word-oriented binary data must be sent to a PDP-11.

Bit 1: Enable data polling

Unless this bit is set, input characters will not be placed in the input FIFO and output DMA transfers will not occur. Unused channels should not be enabled.

Bit 2: Enable interrupt polling

Unless this bit is set, no interrupts will be

requested for this channel. Unused channels should not be enabled.

3.4.7 Command Register

When written into, the low-order 8 bits of this register are stored in the EPCI's Command Register. Reading this register returns a copy of the value last written into it. This register is initialized to zero, and must be set appropriately before the channel will do anything interesting. See the Signetics literature for further information.

Bit 0: Transmitter Enable (TxEN)

When TxEN is set, the EPCI can accept new characters for transmission. If the transmitter is disabled while a character is being sent, it will complete the transmission of the character in the transmit shift register prior to terminating operation. A character in the transmitter holding register will be lost.

Bit 1: Data Terminal Ready (DTR)

Bit 2: Receiver Enable (RxEN)

RxEN enables the receiver portion of the EPCI. Disabling the receiver terminates operation immediately.

Bit 3: Force Break (FB)

Setting this bit causes the transmitter data line to be set to Space after the current character has been completely sent. Normal operation resumes when FB is cleared.

Bit 4: Reset Error (RE)

Setting RE causes the error flags in the status register to be cleared.

Bit 5: Request To Send (RTS)

Bits 7-6: Select operating mode

00 = Normal operation
01 = Automatic echo mode
10 = Local loop back
11 = Remote loop back

3.4.8 Interrupt Enable Mask Registers

These three 16-bit mask registers correspond to the three ID/Vector Registers discussed below. They are used by the BIO microprocessor to determine when to request interrupts. At least once each 250 microseconds, for each mask register in turn, if interrupt polling is enabled (see Mode Control Register B) and no interrupt is already pending, the microprocessor logically ANDs the mask register with the Status Register. If the result of the AND is non-zero, a level 4 interrupt request is asserted, the corresponding ID/Vector is loaded into the Interrupt Acknowledge Data Register (IADR), and the corresponding mask register is cleared.

It is important to note that once a mask register has produced an interrupt, it is cleared and cannot create another interrupt until the software has set it to something other than zero. This is in contrast with machines where once an interrupt is enabled, it remains enabled until explicitly disabled.

3.4.9 Device/Version Number

This read-only register is provided to allow the MC68000 to identify the type of device at this address, and to check the version number of the associated microcode. The 2661 EPCI device as described in this report has been assigned device number zero, and the microcode which corresponds to this report is version 1.0. All modules (like the BIO) which connect to the PNC via the BIOLINK will have Device/Version Number registers at offset 0x1E.

Bits 15-8: Device number: 0 for asynchronous channels

Bits 7-4: PROM number

The BIO microcode is stored in PROM. Major changes or unpatchable bug fixes may require new PROMs to be programmed. At such times the PROM Number will be incremented by one. The current PROM Number is 1.

Bits 3-0: Patch number

Sometimes PROMs can be patched by overwriting. In such cases the next available bit is set in this field. The current Patch Number is 0.

3.4.10 ID/Interrupt Vector Registers

These three 16-bit registers correspond to the three Interrupt Enable Mask Registers discussed above. They are loaded into the IADR by the BIO microprocessor when the corresponding mask register, ANDed with the status register, is non-zero. They have two independent sections.

Bits 7-0: Interrupt Vector (IV)

These 8 bits of vector information are gated to the MC68000's data lines during an interrupt acknowledge sequence. The MC68000's new program counter is fetched from memory at a location (in Segment 0) equal to four times the value of the interrupt vector.

Bits 15-8: Identification number (ID)

When the IADR is passed to the Processor Node Controller (PNC) during a level 4 interrupt acknowledge sequence, the PNC stores the IADR contents into a special location (in Segment F8) reserved for level 4 interrupt acknowledge data words. The ID may be used by the interrupt service routine to discriminate between different EPCI channels or conditions. Thus a single interrupt routine could handle several devices, and determine which one needed service quickly, without testing all of their status registers individually.

3.4.11 DMA Address Register

This 32-bit register contains the local physical address of the next character to be transmitted. It is set up by the MC68000, and incremented by the BIO microprocessor as characters are transmitted. Since only the low word is incremented, blocks cannot cross 64K-byte boundaries. This is not a problem under Chrysalis, since individual objects never cross 64K boundaries. If the channel is in PDP-11 style addressing mode, the low order bit of this address register is complemented when characters are fetched from memory.

3.4.12 Character Block Counter

A DMA transmission is initiated by loading this counter with the number of bytes in a character string to be transmitted. The counter is decremented as each byte is read from memory and loaded into the EPCI. Loading a non-zero value into the counter causes Status Register Bit 8 (TDONE) to be cleared. When the counter reaches zero, the DMA transfer is complete, and TDONE is set. Loading the counter initiates transmission of the first character within 250 microseconds.

3.4.13 Conditions for Normal Operation

The number of stop bits depends upon the setting of the Mode Control register. The MC68000 may specify one, one and a half, or two stop bits; eight bit characters, no parity, and 1.5 stop bits are one reasonable choice. "Mode and Baud Rate Factor" must be set to "Asynchronous 16X rate". Initializing a channel sets the Mode Control Register to a default value of: "16X BKDET async", 9600 baud, 1.5 stop bits, even parity enabled, 7-bit characters, and "Asynchronous 16X rate". The meanings of these bits are explained in Section 4.

The Command Register is set to zero during initialization, i.e., "normal operation" (not auto-echo or looped), set RTS low, "don't clear error flags", "don't force break", disable receiver, set DTR low, disable transmitter. (See the next section for a

discussion of the RTS, DTR, DCD, and DSR modem control signals). The Command Register will have to be used to reenable the 2661; transmitter enable and RTS must be set, and the CTS input must be high, before anything will be transmitted. The receiver requires the receiver enable bit to be set, and the DCD input to be high. For internal loopback, transmitter enable, DTR, and RTS must all be set.

4. Synchronous Channels

The synchronous I/O section of the BIO module meets the strict latency requirements of high speed communications interfaces by handling most of the required I/O processing, control, and data transfer operations with the BIO microprocessor. As a result, the application software only handles complete packets, or batches of packets. This increases overall performance, and decreases the complexity of the application software.

The synchronous section of the BIO module is more sophisticated than the asynchronous section and considerably different from most I/O system architectures. To make things as clear as possible in this report, we have chosen to describe this synchronous I/O system from the point of view of the application programmer, rather than giving a detailed description of the hardware itself. We begin with a description of the line protocols implemented by the system. We follow this with summary descriptions of the two data structures that are fundamental to the operation of the system, and conclude with a detailed explanation of their use.

The synchronous I/O section of the BIO makes extensive use of the Event Mechanism supported by the Butterfly Processor Node Controller. Also, for the sake of clarity, the terms "buffer" and "packet" are given the following definitions: A buffer is a

single block of memory with a header and a data area. A packet is a linked list of one or more buffers.

4.1 Interface Protocols

The signal level protocol implemented by the BIO synchronous I/O channels conforms to EIA standard RS-422. The reader is referred to the relevant EIA documentation for further information.

Protocol processing at the link level on each channel is done by a single LSI chip, the Signetics 2652 Multi-Protocol Communications Controller (MPCC). Two different synchronous signalling protocols are provided by this chip. The Voice Funnel uses the Bit-Oriented Protocol (BOP) described below, which conforms to the framing and message formatting conventions prescribed HDLC. The chip also implements a Byte Control Protocol (BCP). This may also be usable, but we have not investigated running the devices in that mode, and changes to the BIO might be required to do so.

Since these channels will be required to run at up to 2 Mbps, we have been careful to analyze and find ways to meet the maximum service time (latency) constraints implied by the communications protocol chips. The following section discusses BOP and the low-level latency constraints. Later sections discuss how the synchronous channels are controlled, and some

techniques which solve some of the higher-level latency constraints.

4.2 Bit-Oriented Protocol

Bit oriented messages are transmitted in frames. All messages adhere to the standard frame format shown in Figure 4.

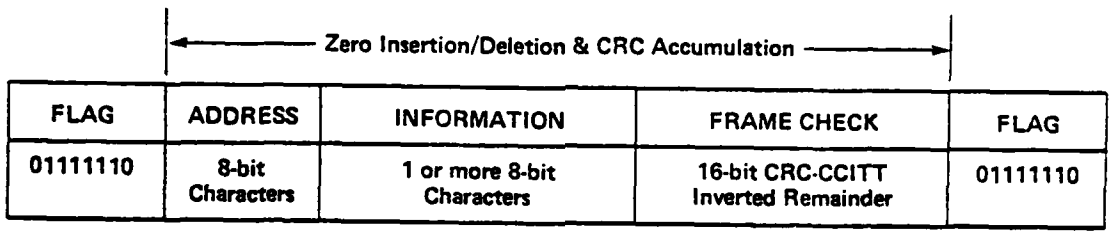

Figure 4 . Bit-Oriented Frame Format

The Bit-Oriented Protocol (BOP) features:

- Independence of codes, line configuration, and peripherals
- Positional significance used instead of control characters or character counts
- Information transparency achieved through zero insertion and deletion
- Error checking on a complete frame.

A frame starts with an 8-bit FLAG sequence, 01111110, followed by an ADDRESS sequence, an INFORMATION sequence, and a FRAME CHECK sequence, and ends with another FLAG sequence.

The BOP is designed for multipoint operation on one data link (i.e., one source of information and one or more destinations). When one station transmits, each station attached to the data link continuously searches for a FLAG sequence and then reads the ADDRESS sequence. The receiving station recognizes its own address and then accepts the remainder of the frame.

The INFORMATION field may vary in length including a different length in sequential frames making up a complete transmission. While the BOP allows the information field to be any number of bits, the BIO hardware limits the information field length to be any non-zero multiple of 8 bits.

Because there is no restriction on the bit patterns that may appear between the end of the start FLAG and the beginning of the end FLAG, the transmitted data stream may contain six or more contiguous ones. This pattern could be interpreted as a FLAG, and inadvertently terminate an incomplete frame. To circumvent this, once the start FLAG has been completed, the transmitting station starts counting the number of contiguous 1's; when five 1's occur, the transmitter automatically inserts a 0 following the fifth 1. The receiver, too, counts the number of contiguous

1's. When the number is five, it inspects the sixth bit; if the sixth bit is 0 the receiving station drops the 0, resets its counter, and continues receiving. But if the sixth bit is a 1, the receiving station continues to receive, interpreting the sequence as a FLAG.

The FRAME CHECK SEQUENCE (FCS) is included in all BOP frames to detect errors which may occur during transmission. This field is 16 bits long and immediately precedes the end-of-frame FLAG. The contents of the FCS field, based on a cyclic redundancy check, is an inverted remainder derived from a division of the transmitted data by a generator polynomial. The dividend is initially preset to 1's, and the data stream that follows becomes the dividend. The generator polynomial for CRC-CCITT is:

$$G(x) = X^{16} + X^{12} + X^{5} + 1.$$

4.3 Channel Control Blocks

The synchronous I/O system maps each packet on the communication interface into one or more buffers in main memory on a Butterfly processor node. Due to the speed of the interface and the architecture of the I/O system, the channel has only a few microseconds to select a new buffer when the end of the current buffer is reached. This is much faster than the application software can respond, so it is impractical to interrupt the MC68000 CPU on a per-buffer basis. In order to get

the MC68000 out of this critical path, each channel interprets a linked list of Channel Control Blocks (CCBs). On input, a CCB serves to identify an empty buffer that is ready to be filled by the Receiver. On output, it serves to identify a buffer that is filled with data and ready for output.

The C structure definition of a Channel Control Block is shown in Figure 5. The functions of the various fields are described in Table 3. Channel Control Blocks belong to the standard set of objects defined by the operating system, and are allocated from supervisor memory. For historical reasons, the structure of a CCB differs somewhat from that of other Chrysalis Objects. In particular, the spare, sequence number, and protection fields common to other types of objects are missing from the Channel Control Block. When constructing an Object Handle, the second most significant byte of the "ccb_phys" field of the CCB is used as the sequence number, since it is in the same position as the sequence number field in other objects.

```
struct ccb
{
    struct ccb      *ccb_next;
    char unsigned    o_type;
    char bits        o_flags;
    physaddr         ccb_phys;
    short unsigned   ccb_nbytes;
    short bits       ccb_status;
    long             ccb_time;
    short unsigned   ccb_size;
}
```

Figure 5 . Channel Control Block Structure Definition

Bit 2:	Post transmitter event
Bit 8:	Start of packet
Bit 9:	End of packet
Bit 10:	Packet terminated by abort
Bit 11:	Packet terminated by Receiver overrun
Bits 14-12:	Number of bits in the last byte (zero implies eight bits)
Bit 15:	Error in the CRC

Table 2. Channel Control Block Status Flags

ccb_next:	Used by the synchronous I/O microcode to follow linked lists of Channel Control Blocks.
o_type:	Used by the operating system to determine what type of object this is.
o_flags:	Used by the object management system.
ccb_phys:	Gives the physical address of the first byte of useful data in the buffer. Used by the microcode to determine the starting address for data transfers.
ccb_nbytes:	Gives the number of data bytes currently in the buffer. For incoming buffers, the microcode fills in the correct value automatically. For outgoing buffers, the microcode uses it to determine how many bytes to transmit.
ccb_status:	The meanings of the bits in this field are summarized in Table 2. On output, only the "Post Transmitter Event" and "End of Packet" flags are meaningful. *On input*, it is filled in by the microcode when it loads the last byte into the buffer.
ccb_time:	On input, this field is filled in by the microcode. It gives the time at which the last byte of data was inserted into the buffer. On output, this field specifies the time at which transmission of the buffer should start. This time is given relative to the real time clock on the I/O board.
ccb_size:	Gives the total number of bytes allocated to the buffer described by this CCB. On input, used by the I/O microcode to decide when the current buffer is full. Not used on output. This field must contain an even number greater than two.

Table 3. Channel Control Block Fields

4.4 Device Control Blocks

The C structure definition of the Device Control Block for a synchronous channel is shown in Figure 6. The function of each field is summarized in Tables 4 through 7. As one might expect, not all of the available addresses in a DCB are used. Control registers that are specific to the transmit side of the channel begin at a byte offset of 0x10 from the base address. Control Registers that are specific to the receive side of the channel begin at a byte offset of 0x104. The first five fields in a Device Control Block are pertinent to both sides of the channel. Writing into any of the undefined fields should be avoided, as many of them are active, and changing their contents could have undesirable side effects.

Note that the "sc_TxCurrent", "sc_RxCurrent", "sc_RxFreeQ" and "sc_RxBufAdd" fields claim to hold pointers, but are only sixteen bits wide. By definition, the Channel Control Blocks that these fields point to must be in Segment F8. Since Segment F8 always starts at the same physical location, and cannot be larger than 64 kilobytes, the width of certain data paths on the I/O board is reduced by treating these fields as offsets.

```
struct synch_chan                                   /* byte offset */
{
    short unsigned sc_enable;                       /* 0x000 */
    short unsigned sc_loop;                         /* 0x002 */
    long           sc_time;                         /* 0x004 */
    long  unsigned sc_err_ev;                       /* 0x008 */
    short unsigned sc_err_code;                     /* 0x00C */
    short unsigned sc_version;                      /* 0x00E */

    short unsigned sc_TxStop;                       /* 0x010 */
    long  unsigned sc_TxEvent;                      /* 0x012 */
    short          sc_TxMindelay;                   /* 0x016 */
    short unsigned sc_TxCurrent;                    /* 0x018 */

    short unsigned sc_GRrw;                         /* 0x01A */
    short unsigned sc_GRadr;                        /* 0x01C */
    short unsigned sc_unused [115];                 /* 0x01E */

    short unsigned sc_RxCurrent;                    /* 0x104 */
    short unsigned sc_RxFreeQ;                      /* 0x106 */
    short          sc_RxFreeCount;                  /* 0x108 */
    short          sc_RxFreeLimit;                  /* 0x10A */
    long  unsigned sc_RxFreeEvent;                  /* 0x10C */

    short unsigned sc_RxBufCount;                   /* 0x110 */
    short unsigned sc_RxBufLimit;                   /* 0x112 */
    short          sc_RxBufLatency;                 /* 0x114 */
    long  unsigned sc_RxBufEvent;                   /* 0x116 */

    long  unsigned sc_RxUnused;                     /* 0x11A */
    short unsigned sc_RxBufAdd;                     /* 0x11E */
}
```

Figure 6 . Synchronous I/O Device Control Block

sc_enable:	Setting and clearing bits in this field independently enable and disable the microcode that services the transmit and receive sides of the channel.
sc_loop:	Setting this field to a nonzero value puts the channel into an internal loopback mode.
sc_time:	A 32-bit real time clock shared by all of the channels on the I/O board. The value of this clock is guaranteed by the operating system software to be within within a few ticks of that of the Processor Node, but not precisely the same.
sc_err_ev:	The physical address of an Event Handle to post when the channel encounters a fatal error condition.
sc_err_code:	When the channel encounters a fatal error condition, it puts an error code in this field to indicate the nature of the condition. Writing to this location causes the I/O microcode to reset both the Transmit and Receive sides of the associated channel.
sc_version	The high byte of this field gives the type of device associated with this Device Control Block, and the low order byte gives the version number of the microcode. As with the Processor Node Controller microcode version number, the low order nibble of the version number distinguishes among patches to the microcode PROMs, and the high order nibble distinguishes among more significant changes. The device type of a synchronous channel is one. The version number of the microcode described in this document is 1.0.

Table 4. Device Control Block Common Fields

sc_TxStop:	Reading this field causes the Transmitter to suspend operation at the end of the current packet. Writing a one into this field causes the Transmitter to resume operation. This field is used to stop and restart the transmitter when a new packet must be added to the Transmit queue.
sc_TxEvent:	The physical address of an event handle to post whenever the Transmitter processes a CCB that has its "Post Transmitter Event" flag set.
sc_TxMindelay:	The minimum number of 62.5 microsecond clock ticks that the transmitter must wait between the end of one packet and the beginning of another. This field is used to keep the transmitter from swamping longer latency devices.
sc_TxCurrent:	A pointer to the Channel Control Block currently being processed by the Transmitter.
sc_GRrw:	See "sc_GRadr" below.
sc_GRadr:	This field is used along with the "sc_GRrw" field to observe and manipulate the internal state of the synchronous I/O microcode for debugging purposes. Writing an address into this field and reading the "sc_GRrw" field reads the current contents of the corresponding microcode register. Writing an address into this field and writing the "sc_GRrw" field loads a value into the corresponding microcode register. Anyone who needs to use these fields should refer to the source code listings for the I/O microcode.

Table 5. DCB Transmitter and Debugging Fields

sc_unused:	An array of locations not used by the application software. These locations should not be written to, as many of them are active, and changing their contents could have undesirable side effects.
sc_RxCurrent:	A pointer to the Channel Control Block that is currently being processed by the Receiver.
sc_RxFreeQ:	The sixteen bit supervisor memory address of a control block that gives access to a queue of "auxiliary CCBs". This queue is used when there are no empty buffers left on the Receiver queue. A value of zero in this field indicates to the microcode that there is no such queue available.
sc_RxFreeCount:	This field is decremented each time the Receiver finishes processing a buffer. Whenever the value of this field falls below the value of the "sc_RxFreeLimit" field, the "sc_RxFreeEvent" is posted. For indivisibility reasons, writing the value N into this field actually adds N to its contents.
sc_RxFreeLimit:	The "sc_RxFreeEvent" is posted whenever the value of the "sc_RxFreeCount" field falls below the value of this field.
sc_RxFreeEvent:	The physical address of an event handle to post when the value of the "sc_RxFreeCount" field falls below the value of the "sc_RxFreeLimit" field.

Table 6. Device Control Block Receiver Fields

sc_RxBufCount:	This field is incremented each time the Receiver finishes processing a buffer. Whenever the value of this field gets to be equal to the value of the "sc_RxBufLimit" field, the "sc_RxBufEvent" is posted. For indivisibility reasons, reading this register also sets it to zero.
sc_RxBufLimit:	The "sc_RxBufEvent" is posted whenever the value of the "sc_RxBufCount" field gets to be equal to the value in this field.
sc_RxBufEvent:	The physical address of an event handle to post when the value of the "sc_RxBufCount" field gets to be equal to the value of the "sc_RxBufLimit" field.
sc_RxBufLatency:	The maximum time that the Receiver is allowed between the end of a packet and the posting of the "sc_RxBufEvent". This protects against the case where the Receiver has processed a packet that is not large enough to trigger the "sc_RxBufEvent".
sc_RxUnused:	Not used by application software
sc_RxBufAdd:	Writing the low order sixteen bits of the supervisor Memory address of a Channel Control Block into this field will cause the microcode to splice the CCB into the Receiver queue. This is used to avoid race conditions when an application program adds empty buffers to the Receiver queue. For reasons of simplicity, the microcode splices the new CCB into the queue directly behind the CCB that is currently being processed. When this field is read, it returns a nonzero value if the microcode has not yet completed the last splicing operation, and zero otherwise. The MC68000 should always check before writing to this field.

Table 7. DCB Receiver Control Fields (continued)

4.5 Enabling the Channel

The "cs_enable" field of the Device Control Block controls whether or not the microcode polls the communications protocol chips. In order for data to be transfered between the protocol chip and memory, these bits must be set. Table 8 gives the use of these bits.

Bit 0: Unused

Bit 1: Enable Receiver microcode polling.

Bit 2: Enable Transmitter microcode polling.

Bits 15-3: Unused

Table 8. Valid Settings for the Field cs_enable

The purpose of this control register is to conserve the bandwidth of the BIO micromachine. The BIO micromachine does not have enough bandwidth to support all of the I/O devices if they are operating at full speed. Therefore, if some lines are unused, they should not be enabled.

Note that this field does not affect the communications protocol chip in any way. In particular, the receive side of the chip is ready to start filling the input ring buffer immediately after reset, and will overrun if the receiver is not enabled soon enough. When the transmitter enable bit is cleared, the Transmitter microcode polling will stop transferring data from

main memory only when it reaches the end of the current output packet. Similarly, clearing the receiver enable bit will cause the Receiver microcode to stop transferring data into main memory when it reaches the end of the current input packet.

4.6 Looping the Channel

Writing a nonzero value into the "sc_loop" field connects the transmit channel output to the receive channel input. An external clock input to the transmit channel is still required, but the receiver clock input is disabled when in this mode. The loopback connection is internal to the communications protocol chip, so it does not exercise the TTL drivers that bring the signals off the chip, or the circuits that drive the I/O cables. Since both of these are common failure points, external looping of the channel at or beyond the drivers is recommended when testing.

4.7 Fatal Errors

When the synchronous I/O hardware encounters a fatal error condition, it puts an error code in the "sc_err_code" field, posts an event, and shuts down both its transmitter and its receiver. The meanings of the error codes are shown in Table 9. To post the event, the I/O microcode retrieves an event handle from the physical address specified in the "sc_err_ev" field, and

passes it to the Processor Node Controller. All of the events posted by the Synchronous I/O system are posted in this way. After posting the error event, the channel goes into a quiescent state. It does not reset itself at this point, as that would clear all information pertinent to the error condition.

Code	Meaning
1	The Transmitter has encountered a CCB with a null link pointer that was not at the end of a message.
2	Transmitter Underrun. This occurs when the I/O micromachine is so heavily loaded that it cannot transfer data from its ring buffer to the communications protocol chip fast enough.
3	Because of the way that it tests for end conditions, the Transmitter microcode cannot handle a packet if the value of its "ccb_nbytes" field of a CCB on is greater than or equal to 32 kilobytes. It sets this error code whenever that condition occurs.
4	The channel was unable to post the "sc_TxEvent".
5	The channel was unable to post the "sc_RxFreeEvent".
8	The channel was unable to post the "sc_RxBufEvent".

Table 9. Synchronous Channel Error Codes

As indicated in the table, it is a fatal error if any event cannot be posted by the channel. This failure can be due to one of two conditions: (1) invalid Event Handle, or (^) an attempt to multiply post the handle of an event block that has its "illegal to post multiply" flag set. If the posting of the error event fails, the channel enters its quiescent state without taking any further action.

4.8 The Reset Function

The hardware and microcode state variables associated with a channel may be reset by writing to the "sc_err_code" field of its Device Control Block. As noted below, the Reset Function causes the Transmitter and Receiver to lose all information about the CCB lists that they are processing. These lists must be reinitialized whenever this function is invoked.

The procedure for shutting a synchronous channel down is summarized in Table 10. Since the operation of the synchronous I/O section is rather complex, it is not surprising that the procedure for shutting a channel down without undesired side effects is also complicated. The first step in the procedure is to turn off the microcode polling function that reads and writes Channel Control Blocks. Since the guaranteed latency of the polling function is 250 microseconds, the polling function will see the transition of the enable flags and shut down within that period.

1. Clear the "sc_enable" field of the DCB.
2. Wait 250 microseconds.
3. Inhibit interrupts to the MC68000.
4. Write to the "sc_err_code" field of the DCB.
5. Wait 100 microseconds.
6. Write to the "sc_err_code" field of the DCB.
7. Wait 100 microseconds.
8. Enable interrupts to the MC68000.

Table 10. Procedure for Resetting a Synchronous Channel

The reset request may interrupt an ongoing DMA transfer related to the channel in question. The reset function clears the DMA word count, so this operation will terminate immediately when control is returned to it, but a second reset operation is necessary to clear the state variables that are affected when the DMA routine makes its exit.

The waiting period after each of the reset operations serves two purposes. First, the microcode will ignore the second invocation of the reset function if it is made before the first reset is complete. Second, the reset function ties up the entire I/O micromachine (asynchronous and synchronous channels) uninterruptibly for approximately 25 microseconds (150 microcycles). The waiting period gives the micromachine a chance to catch up on other business before the MC68000 proceeds.

The need to inhibit interrupts to the MC68000 is also due to the fact that the reset function ties up the micromachine for a

relatively long time. If interrupts were not inhibited, the MC68000 could enter an interrupt routine of its own just before the execution of either one of reset functions. If this interrupt routine sets up a DMA transfer to an Asynchronous I/O channel, for instance, the Processor Node Controller could end up making a request of its own to the same BIO module. The combination of the long execution time of the reset function and the potentially heavy processing load on the I/O micromachine on completion could cause it to fail to respond to the PNC in time, causing the PNC to signal a spurious bus error, or worse.

4.9 Real Time Clock

The "sc_time" field provides exactly the same facility as the real time clock register found in memory subspace zero of all Butterfly Processor nodes. That is, it is a 32 bit read/write register whose value is incremented once every 62.5 microseconds. When specifying times relative to this clock, it is important to remember that it counts modulo 32 bits. The range of times relative to any specific value of this clock is therefore plus or minus 2**31 units, or half the 74+ hour range of the clock.

Any program with access to the Device Control for a synchronous channel can reset the I/O board clock by simply writing the "sc_time" register. When writing this register, it is necessary to take the same precautions observed when setting

the Processor Node Controller real time clock. Since the I/O board real time clock is shared by all of the synchronous channels on the BIO, setting the clock through any one of the Device Control Blocks will change its value for all of the synchronous channels on the BIO.

When a Butterfly processor is reset, all of the Processor Node clocks are cleared, but none of the I/O board clocks are cleared. This means that in general, the two sets of clocks will not be synchronized. When the Chrysalis Operating System starts up, it sets all of the I/O board clocks under its control to be as close as possible. In fact, they will often be exactly synchronized, but it is impossible for the operating system software to guarantee this.

4.10 Transmitter

The synchronous Transmitter is implemented as a pair of microcode processes. One of these processes reads characters from a 64 byte ring buffer, feeds them to the communication protocol chip as it requests them, and takes care of start and end of packet conditions. The other process reads characters from main memory and puts them into the ring buffer. The microcode moves down a linked list of Channel Control Blocks, using each one as a specification of what is to be done with a particular block of memory. This linked list of CCBs is referred

to here and in other documents as the "Transmit Queue". The detailed operation the Transmitter microcode is outlined in the pseudo program at the end of this section. At the interface to the application programs, it implements the following loop:

1. Follow the "ccb_next" field of the current CCB to find the next CCB in the Transmit Queue.
2. Post an error event if the value of the "ccb_nbytes" field is greater than 0x7FFF or equal to zero.
3. Starting at the address specified by the "ccb_phys" field of the CCB, read data into the Transmitter ring buffer from main memory.
4. Keep reading data as fast as the ring buffer allows, until the number of bytes read equals the value of the "ccb_nbytes" field of the CCB.

This is illustrated graphically in Figure 7. Here, Buffer 2 is being processed, as indicated by the "sc_TxCurrent" pointer, while Buffer 1 has been processed and is waiting to be recycled, and Buffer 3 is waiting to be processed. When the channel reaches the end of the Transmit Queue it enters a quiescent state, in which it simply marks time.

There are some additional features of the Transmitter microcode that complicate the basic loop shown above, and give it some added flexibility. These include actions triggered by flags in the "ccb_status" field of a CCB, maintaining a minimum time interval between packets, and a provision for specifying the precise transmit time of a packet. These are discussed in the

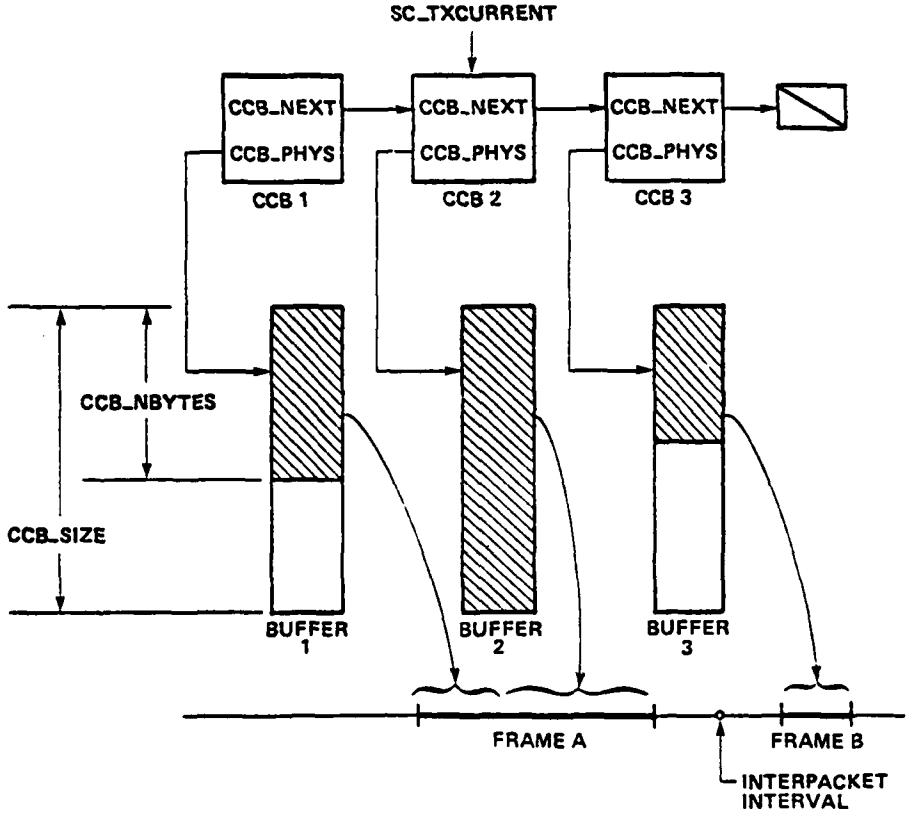

Figure 7 . Output of Three Buffers Into Two HDLC Frames

following subsections.

4.10.1 Channel Control Block Flags

The Transmitter pays attention to two flags in the "ccb_status" field of a Channel Control Block. If the "Post Transmitter Event" (0x0004) flag is set, the microcode posts the

Event Handle found at the physical address stored in the "sc_TxEvent" field of the Device Control Block. This serves to notify the application program that one or more buffers have been processed and are ready to be freed from the head of the Transmit Queue. If the "end of packet" (0x0200) flag is set, the current packet will be terminated after the last byte in the current buffer has been transmitted. This involves telling the communications protocol chip to send its CRC word and an HDLC terminating flag, and setting up for the beginning of the next packet.

4.10.2 Packet Transmit Times

The Transmitter will not start to transmit a new packet until the value of the I/O board real time clock is greater than or equal to the value in the "ccb_time" field in the last CCB of the previous packet. This allows the application program to set the precise time (within 62.5 microseconds) at which a packet will be transmitted. This is useful in applications such as satellite host stream processing in the Voice Funnel, where message bursts happen at tightly constrained intervals. While having the last CCB of a packet control the transmit time of the following packet is not the most obvious choice, it turns out to provide the simplest and most flexible user interface.

When specifying a packet transmission time in the "ccb_time" field of a CCB, it is important to recall that the I/O board real time clock and its counterpart on the processor node have the same period, but not necessarily the same value. The value in the "ccb_time" field should be specified relative to the I/O board clock. If the precise transmit time is important, the application code must ensure that packets on the Transmit queue remain sorted with respect to transmit time. If the precise transmit time is unimportant, the application code should store the current value of the I/O board clock in the "ccb_time" field of the last CCB in every packet, since there is no way to disable this mechanism.

The minimum time interval between the end of one packet and the beginning of the next is controlled by the "sc_TxMindelay" field of the Device Control Block. The Transmitter will not start to transmit a new packet until the number of clock ticks that have elapsed since the end of the last packet is greater than or equal to the value in the "sc_TxMindelay" register of the Device Control Block. This keeps the I/O board from swamping other devices with longer latencies without constraining the Butterfly application software. The microcode treats this field as a signed quantity, so the largest interpacket interval allowed is 32,768 clock ticks, or 2.048 seconds. Specifying a negative interval is equivalent to specifying a zero length interval.

There are two minor subtleties worth noting here. First, there is a difference between loading a packet into the Transmitter ring buffer and transferring characters from the ring buffer to the communications protocol chip. The former commences as soon as the Channel Control Block is read, while initiation of the latter is subject to the interpacket spacing and transmit time parameters discussed above. Second, if the Transmitter's ring buffer happens to be empty when a new packet is queued for transmission, the firmware will not start transferring characters to the communications protocol chip until the ring buffer is full. This has the beneficial effect of minimizing the chance of a transmitter underrun condition. It also has the slightly undesirable effect of delaying the transmission of packets that arrive under these conditions by approximately 32 to 50 microseconds.

4.10.3 Maintaining the Transmit Queue

Before it tries to splice a new packet into the Transmit Queue, the MC68000 must stop the the Transmitter in order to avoid a race condition with the microcode. It does this by reading the "sc_TxStop" field of the Device Control Block. When this field is read, the microcode first returns the low order sixteen bits of a pointer to the CCB that is currently being processed by the Transmitter. The transmitter stops when it finishes transmitting the last byte of the current packet. In

the meantime, the MC68000 is free to change the contents of any CCB on the transmit queue that follows the last CCB in the current packet. It is also allowed to change the "ccb_next" field of the last CCB in the current packet, but no other field in that CCB. When it is done rearranging the Transmit queue, the MC68000 restarts the channel by storing a value of one into the "sc_TxStop" field.

When the transmitter is sending a multi-buffer packet, the value returned when the "sc_TxStop" field is read will not necessarily be pointing to the last CCB processed by the microcode before it stops. Before any manipulations on the Transmit Queue are performed it is necessary to find the end of the current packet by starting with the the pointer returned from the "sc_TxStop" field and following the transmit queue until a CCB is found with its "end of packet" bit set.

Note that it is necessary to stop the transmitter only when CCBs are added to the Transmit Queue "downstream" from the CCB that is currently being processed. Processed CCBs can be removed from the head of the Transmit Queue for recycling at any time without stopping the Transmitter.

4.1C.4 Initialization

The Transmitter must have at least one Channel Control Block on its queue at all times. Thus, in the limiting case, the

Transmit queue consists of a single CCB with a null "ccb_next" field, pointed to by the "sc_TxCurrent" field of the Device Control Block. As a result, reinitialization of the Transmitter involves more than just invoking the reset function. After the channel has been reset and before the Transmitter is enabled, the MC68000 must store a pointer to a null terminated list of CCBs into the "sc_TxCurrent" field of the Device Control Block. The first CCB in the list is never processed. It can be recovered as soon as the Transmitter goes on to process another CCB.

Note that the minimum number of buffers needed to keep a Transmitter from blocking is two instead of one, since the last CCB on the Transmit Queue is never freed.

4.11 Receiver

The basic structure of the synchronous Receiver is similar to that of the Transmitter. Like the Transmitter, it is implemented as a pair of firmware processes. One process reads characters from the communications protocol chip and puts them into a 64 character ring buffer, while the other takes characters from the buffer and reads them into main memory on the local processor node. Also like the Transmitter, the microcode walks down a linked list of Channel Control Blocks. It uses each CCB first as a specification for a block of memory into which it is allowed to transfer incoming data, and then as a place to record

information about the data that it has transferred. This list of CCBs is referred to here and in other documents as the "Receive Queue". The detailed operation of the microcode is outlined in the pseudo-C program shown at the end of this section. At the user interface, it implements the following loop:

1. Follow the "ccb_next" field of the current CCB to find the next CCB in the Receive Queue.
2. Start reading bytes from the Receiver ring buffer into the main memory of the processor node starting at the address specified by the "ccb_phys" field of the CCB.
3. Keep reading bytes as fast as the ring buffer allows, until the number of bytes read equals the value of the "ccb_size" field of the CCB, or until the packet must be terminated due to an end of packet flag or an error condition.
4. Update the "ccb_status", "ccb_nbytes", and "ccb_time" fields of the CCB.

The operation of the Receiver is shown graphically in Figure 8. Here, Buffers 1 and 2 each contain part of a single packet, while buffer 3 contains an entire packet. Bytes from the current packet are being transferred into buffer 4, as indicated by the "sc_RxCurrent" pointer. If the Receiver gets to the end of buffer 4 before the end of the packet, it will move on to buffer 5.

The firmware actually transfers 16 bits of information each time it moves a character from the communications protocol chip. The extra eight bits are the contents of a status register in the protocol chip at the time that the character was read. This

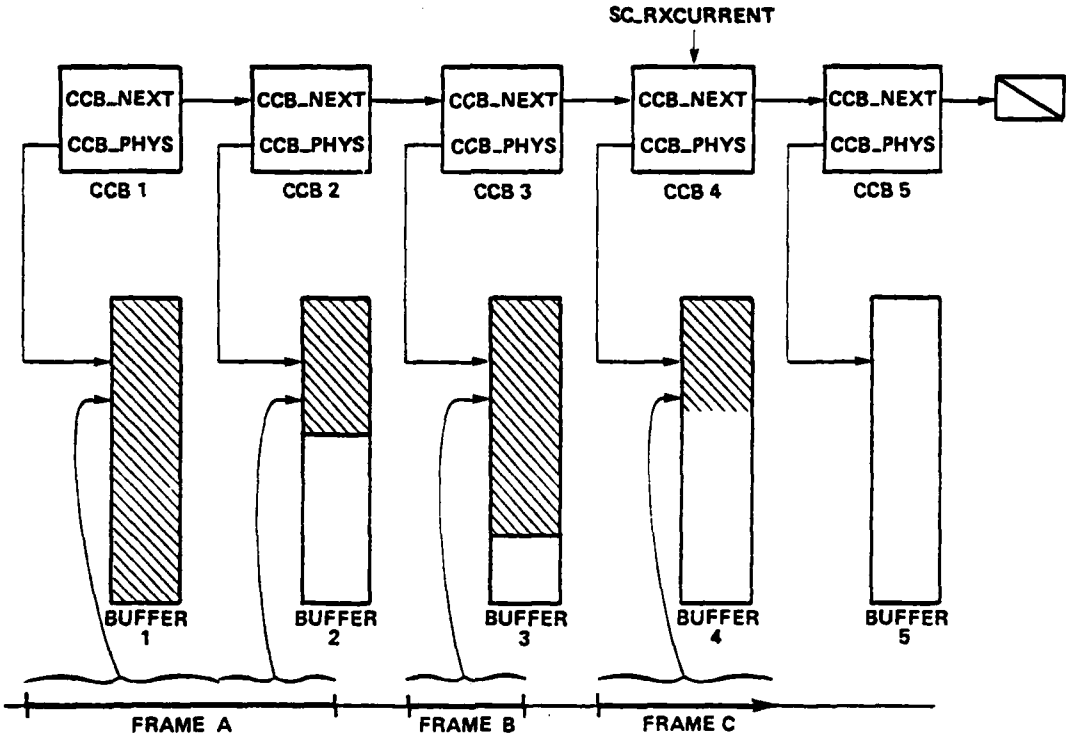

Figure 8 . Input of Three HDLC frames into five buffers

information is kept in the Receiver ring buffer along with the incoming character, and is processed as the character is transferred to the main memory of the processor node. If either "End of Packet", "Abort", or "Receiver Overrun" is set in this field, the current packet is terminated. We refer to an occurrence of any of these conditions as an "end of packet condition".

The microcode will move on to the next Channel Control Block in the Receive Queue when the current buffer is full, or when an end of packet condition occurs. Before it moves on, the microcode updates the current CCB in the following ways:

ccb_nbytes: The number of data bytes actually copied into the buffer is entered into this field.

ccb_status: The contents of the status register of the communications protocol chip when the last byte in the buffer was received is copied into the high order byte of this field. The value of the last byte in the buffer is copied into the low order byte.

ccb_time: The value of the I/O board real time clock when the last byte was copied into the buffer is stored in this field. This may be a few clock ticks later than the time at which the last byte was actually received, since incoming characters must percolate through the Receiver ring buffer.

In general, the "ccb_status" field is only useful at the end of a packet. In that case, the bits in this field are interpreted as follows:

Bits 7-0: The value of the last byte in the packet.

Bit 8: This buffer contains the first byte of a packet.

Bit 9: This buffer contains the last byte of a packet.

Bit 10: This buffer contains the last byte of a packet that was terminated by Abort Flag.

Bit 11: This buffer contains the last byte of a packet that was terminated by a receiver overrun condition.

Bits 14-12: Since the line protocol is bit oriented, it is possible to receive a packet that does not end on a

byte boundary. When this is the case, the three bits in this field give the number of bits in the incomplete byte at the end of the packet.

Bit 15: A CRC error was detected in the packet whose last byte is in this buffer. This flag can be set only if the end of packet flag is set.

As with the Transmitter, there are certain features that complicate the basic Receiver loop and lighten the load on the application program. These features have to do with the manner in which the channel acquires empty buffers, and the mechanisms used to inform the application program that new data has arrived. These are discussed in the following subsections.

4.11.1 Backup Buffers

The "sc_RxFreeQ" field in the Device Control Block gives the Receiver a backup source of free buffers in the event that the Receive Queue runs out of buffers. This backup source, referred to here and in other documents as the "PNC Queue", is maintained by the application code using the indivisible Supervisor Memory enqueue and dequeue operations available from the Processor Node Controller (these are not the same as the Dual Queue operations supported by the PNC). When the Receiver gets to the end of its queue, it looks in the "sc_RxFreeQ" field for a pointer to a parameter block in Supervisor Memory. It uses this parameter block to invoke the PNC dequeue function. If the dequeue is successful, the PNC will return a pointer to a Channel Control

Block, and the Receiver can proceed. By zeroing the "sc_RxFreeQ" field, the application program can indicate to the Receiver that there is no PNC Queue available.

This function is intended to provide support for a buffer allocation scheme similar to those used in the IMP program and elsewhere. Since the PNC Queue can be shared among several synchronous receivers, it is possible to allocate just enough buffers to each receiver to ensure that lockup conditions will never occur, then leave the remaining buffers on a PNC Queue to take care of bursts when and where they occur. Further discussion of this kind of buffer management strategy may be found in [Rosen, E., "Issues in Buffer Management" Internet Experiment note #182, May 1980.] which treats the subject of buffer management in communications processors in more detail.

When the Receiver has exhausted all potential sources of buffers, it simply stops transferring bytes from its ring buffer to main memory. If the ring buffer fills up and new data arrives, the chip will signal an overrun condition, which is cleared as soon as the microcode starts reading characters again.

4.11.2 Counting Free Buffers

Under normal conditions in a well designed application, the mechanisms described so far are sufficient to keep a Receive channel functioning smoothly. However, there will always be

circumstances when a channel will run short of buffers, at least temporarily. The "sc_RxFreeCount", "sc_RxFreeLimit" and "sc_RxFreeEvent" fields in the Device Control Block provide a flexible, low-overhead mechanism for notifying the application software when this occurs.

The basic rules are that the sc_RxFreecount register is decremented each time a Channel Control Block is processed by the Receiver, and the Event "sc_RxFreeEvent" is posted whenever the value in the "sc_RxFreeCount" field drops below the value in the "sc_RxFreeLimit" field. The intent is that the "sc_RxFreeCount" field contain a count of the number of empty buffers available to the Receiver, and that the "sc_RxFreeLimit" field indicate a threshold below which the Receiver is considered to be abnormally low on buffers. When the "sc_RxFreeEevent" is posted, the application code may simply make a note of the fact that the condition occurred, or it may attempt to take corrective action. The phrase "abnormally low on buffers" is deliberately vague, since the number of buffers available to a channel depends on the size of its Receive Queue, the size of the PNC queue, if any, and the number of Receivers sharing the PNC queue.

To make this work, the application code must increment the "sc_RxFreeCount" field each time it adds an empty buffer to the Receive Queue. It must also initialize the "sc_RxFreeCount" and the "sc_RxFreeLimit" fields to appropriate values before it starts up the Receiver. Since these fields do not influence the

behavior of the Receiver in any way other than to cause it to post an event under certain conditions, their absolute values are not important. It is only necessary to set the difference between the two. Since the microcode treats these fields as unsigned sixteen bit integers when it is compares them, it is possible to disable this mechanism by setting the "sc_RxFreeLimit" field to zero.

In order to avoid race conditions between the microcode and the application code, writing the value N to the "sc_RxFreeCount" field actually increments it by that amount. This makes it possible to increment the field reliably without stopping the Receiver. Instead, it is necessary to stop the Receiver in order to initialize the field.

4.11.3 Signalling Packet Arrival

As packets arrive, the Receiver must let the application software know that there is new work to be done. The microcode provides several facilities to help the application software keep track of incoming packets with a minimum of effort.

The first and most basic facility is a buffer counter. Each time the Receiver processes a CCB, it increments the "sc_RxBufCount" field of the Device Control Block. This lets the Receiver maintain an accurate running count of the number of buffers ready for processing without scanning the Receive Queue

repeatedly or waking up every time a new buffer arrives. When the application code reads the "sc_RxBufCount" field, the microcode automatically zeros it. Combining the read and clear into a single indivisible operation avoids race conditions with the microcode. By periodically adding the contents of the "sc_RxBufCount" field to an auxiliary variable, and decrementing the auxiliary variable whenever a received buffer is processed, the application software can maintain an accurate count of buffers to be processed with relatively little effort.

The method by which the application software is actually notified of the arrival of new data is similar to the method by which it is told that the channel is low on buffers. Like the "sc_RxFreeLimit" field, the "sc_RxBufLimit" field determines a threshold. When the value of the "sc_RxBufcount" field gets to be equal to the value of the "sc_RxBufLimit" field, the "sc_RxBufEvent" is posted. In the simplest case, the "sc_RxBufLimit" field is set to one. Each time a new buffer arrives, the application software receives the event, resets the "sc_RxBufCount" field and processes a buffer. If it is more convenient to process buffers in batches, the value of the "sc_RxBufLimit" field can be set to a larger value.

Setting the "sc_RxBufLimit" field to a large value can result in unacceptably long latency for isolated packets that happen to be too short to trigger the "sc_RxBufEvent". The "sc_RxBufLatency" field is used to solve this problem. This

field sets the maximum number of clock ticks that may elapse between the end of any packet and the posting of the "sc_RxBufEvent".

For the purposes of this explanation, only the simplest possible uses for these fields have been described. There are less obvious ways to use them that might be more useful under some circumstances. When both the "sc_RxBufLimit" and "sc_RxBufLatency" fields are set to zero, for instance, the "sc_RxBufEvent" will be posted once at the end of each incoming packet, regardless of packet length. If the "sc_RxBufLimit" field is set to one and the "sc_RxBufLatency" field is set to zero, the "sc_RxBufEvent" will to posted once at the beginning of each packet, so the application can get going on header processing right away, and again at the end of each packet, so the software can find out when it has a complete packet without scanning the Receive Queue repeatedly.

4.11.4 Maintaining the Receiver Queue

When the MC68000 is splicing new CCBs into the Receive Queue, it must avoid race conditions with the microcode. Unlike the Transmitter, the Receiver cannot be stopped without raising the possibility of an overrun condition. On the other hand, it is not necessary to splice multi-buffer packets onto the Receive Queue, or maintain any kind of ordering among CCBs on the queue.

It is therefore easy for the microcode to support the splicing operation. When an application program wishes to splice a CCB into the Receive Queue, it simply writes the low order sixteen bits of a pointer to the CCB into the "sc_RxBufAdd" field of the Device Control Block. In order to keep this operation as simple and efficient as possible, the microcode splices each new CCB into the queue immediately behind the CCB that is currently being processed.

The "sc_RxBufAdd" field is readable as well as writable. If it is read while the microcode is in the process of splicing a new buffer into the queue, a nonzero value will be returned. Otherwise, this field contains a zero. If the MC68000 attempts to load the "sc_RxBufAdd" field while the microcode is in the process of splicing another CCB into the queue, a fatal error condition can result. It is therefore important to check the value in the "sc_RxBufAdd" field before storing a pointer to a new CCB into it. This is not meant to imply that the microcode splicing operation is slow. In all but a few cases, it should be possible for the MC68000 to add CCBs to the Receive Queue as fast as it can find them. In the cases where the MC68000 must wait, the delay will be very short.

4.11.5 Initialization

The Receiver must have at least one Channel Control Block on its queue at all times. Thus, in the limiting case, the Receive queue consists of a single CCB with a null "ccb_next" field, pointed to by the "sc_RxCurrent" field of the Device Control Block. As a result, reinitialization of the Receiver involves more than just invoking the reset function. After the channel has been reset and before the Receiver is enabled, the MC68000 must store a pointer to a null terminated list of CCBs into the "sc_RxCurrent" field of the Device Control Block. The first CCB in the list is never processed. It is can be recovered as soon as the Receiver goes on to process another CCB.

Note that the minimum number of buffers needed to keep a Receiver from blocking is two instead of one, since the last CCB on the Receive Queue is never freed.

4.12 Byte Ordering

Although communication on the synchronous channels is byte oriented, The Transmitter and Receiver both reference their buffers one word at a time, starting at the word with the lowest address. Each word is emptied (and filled) starting at the low order bit. Note that this runs somewhat counter to the byte addressing conventions of the Motorola MC68000 (which, unfortunately, are not consistent with its bit ordering

conventions). The single exception to this rule occurs when a packet contains an odd number of bytes and begins at an even address. In this case, the low order byte (the high order eight bits) of the last word in the buffer will be used by the transmitter and receiver as the source/destination for the last byte of data. This makes it possible to transfer byte-oriented messages between the Butterfly and other machines with the same byte ordering conventions. For byte oriented transfers between the Butterfly and machines with different addressing conventions (e.g. a PDP-11), byte swapping is necessary.

The only situation encountered so far in the Voice Funnel application where the byte ordering conventions run counter to this scheme is the Lexnet protocol, where the formats of the source and destination addresses follow the conventions of the PDP-11 architecture. Since these headers are quite short, this does not pose a problem. All of the other protocols used in the Wideband Network, including the DOD standard protocols, are word oriented, so byte ordering conventions are not a concern.

5. Appendix: Algorithms Used in the BIO

This appendix contains details of the actual microcode algorithms used in the BIO. Since these tables were extracted from the comments in the actual microcode, they may be difficult to follow, but in exchange, they describe the actual implementation.

Transmitter Microcode State Variables

sofNb — Number of bytes not yet loaded into the output fifo from the data block in the Processor Node's local memory.

TxBOM — Number of beginning of messages loaded into the output fifo minus the number of messages started. TxBOM is used by the output fifo unload process to determine when to begin transmission of the oldest message.

TxBOM	Output fifo contents
0	No unstarted messages
1	One message which has not been started
2	One entire message plus beginning of second message

sOFrd — Remaining number of real time clock intervals in intermessage delay

soAh
soAl — High & low 16-bits of physical address of next memory location to access for loading the output fifo.

uOFs — State of unloading the output fifo:

-1=> waiting for output fifo to be filled with data from 68000 local memory
0=> data from output fifo currently being loaded into the 2652's transmitter output register and waiting for the end of message to be detected
1=> end of message delay has not been satisfied and waiting for the advancement of the real time clock to satisfy the intermessage delay

lOFs — State of loading the output fifo:

-1=> output fifo is currently being loaded with data from 68000 local memory
0=> one of the conditions necessary to begin loading the output fifo with a new message is not satisfied.
1=> the transmitter section is stopped

sTmode — State of transmitter operation

-1=> running
0=> stop at end of current message (set by stop command)
1=> continue transmission of messages (set by unstop command)

Transmitter Microcode Pseudo-code Specification

```
if(lOFs=-1 & sofNb <= 0)
  ;Here last byte was transferred from current 68000 buffer to
  ;output fifo
  if(current control block does not have END_OF_MESSAGE bit set)
    ;Here next control block references a buffer containing more
    ;of the message
    fetch pointer to next channel control block
    if(pointer to next CCB is null)
      execute fatal error post to 68000 [error 1]
      close down transmitter channel
      set smcBr=0
     .END
    else
      ;Here next CCB exists
      fetch parameters in next CCB
      make next CCB the current CCB
      if(length is less than one)
        ;Here secondary data buffer is too short
        execute fatal error post to 68000 [error 3]
       .END
      enable data transfer from new 68000 buffer to output fifo
      go to the lOFs=-1 code above [because the buffer may be very short
     .END
   .END
  else
    ;Terminate current message in buffer.
    load message terminating entries into the output fifo
    if(POST_MESSAGE bit is set)
      post 68000 for successful transmission of message
     .END
    set lOFs=1          ;get a new buffer
    go to "lOFs.1.new" below
   .END
 .END
```

Transmitter Microcode Pseudo-code Specification (cont)

```
if(lOFs=0)
  ;Here a new message is to be loaded into the output fifo, if its
  ;time of transmission condition is satisfied
  if(sTmode != -1)      ;ie, 68000 stop or continue
    set lOFs=1
  else if(transmission time of next CCB <= time of day)
    if(sTmode = -1)     ;retest non-interruptably
      ;Here the next message can begin being loaded into the output fifo
      make next CCB the current CCB
      increment TxBOM
      load message start data into the output fifo
      fetch parameters of current CCB
      set lOFs=-1
      if(start address is odd)
        insert first data byte in fifo
        decrement sofNb
        if(length was 1 byte)
          goto the lOFs=-1 code
       .END
      if(length was less than one)
        ;Here primary data buffer is too short
        execute fatal error post to 68000 [error 3]
       .END
      ;enable data transfer from new 68000 buffer to output fifo
      ;at this point the interrupt routine sofL takes over, and
      ;runs until the buffer is completely loaded or the fifo is full
      go to the lOFs=-1 code above [because the buffer may be very short]
     .END
   .END
 .END


if(lOFs=1)
  ;Here we are stopped between messages
  if(sTmode=1)          ;68000 wants us to continue
lOFs.1.new:     ;enter here from lOFs=-1 code
    if(sTmode != 0)     ;retest uninterruptably
      set sTmode=-1
      fetch pointer to next CCB from current CCB
      if(pointer to next CCB is not null)
        fetch transmission time of next CCB from current CCB
        set lOFs=0
     .END
   .END
 .END
```

Transmitter Microcode Pseudo-code Specification (cont)

```
if(uOFs=1 and intermessage delay condition satisfied)
  set uOFs=-1
 .END

if(uOFs=-1)
  if(TxBOM is greater than zero)
    ;Here transmission of a new message can begin [since sofL has run]
    set uOFs=0
    enable transfer of output fifo data to 2652
    decrement TxBOM
   .END
 .END

if(uOFs=0)
  ;Here message data is being transferred from the output fifo to the 2652
 .END
```

Receiver Microcode State Variables

sifNb — Number of bytes not yet loaded from the input fifo into the data block in the Processor Node's local memory, minus two.

siAh
siAl — High & low 16-bits of physical address of next memory location to access for unloading the input fifo.

uIFs — state of unloading the input fifo NOTE: the values 0,2,4,6 represent states 0,1,2,3!

- 0: acquire an input buffer; when successful go to state 1 or 2.
- 1: Input fifo data is currently being loaded into 68000 local memory and the timer is running; we are waiting for the timer to finish, or dma to complete.
- 2: Input fifo data is currently being loaded into 68000 local memory; we are waiting for last byte in block to be transferred, or for EOM or error.
- 3: Entire buffer has been loaded into local memory; process it, then go to state 0. Set timer if buf_limit is reached, or on first EOM or error.

Receiver Microcode pseudo-C Specification

```
Sin(ch)     /* ch is the channel number, handled by hardware */
{
  while (sIfreeB != 0) {     /* if there is a ccb to free, */
    sIfreeB->spb_link = CCB->spb_link; /* append rest of free list to it */
    CCB->spb_link = sIfreeB;           /* make it first on the new list */
    sIfreeB = 0;                       /* prepare to take another ccb */
    }
  switch(uIFs) {          /* dispatch on state */
    case(3):              /* sifU sets state=3 when a buffer is complete */
      CCB->timestamp = time_of_day;
      CCB->status = status[sIFeA-1];            /* status of last byte */
      CCB->actual_len = (CCB->max_len - 2) - sifNb;

      if ((status & (REOM+RAB+ROR)) != 0)       /* if end-of-message */
      if (sItime == 0)                        /* and timer not set */
        sItime = (time_of_day-1 + sIwait) | 1;  /* time to do post*/
      sIbufC += 1;                           /* inhibit microinterrupts */
      if (sIbufC == sIbufL)                  /* buf_count == buf_limit? */
        sItime = (time_of_day-1) | 1;        /* do post(data) now */

      uIFs = 0;
      /* fall thru into case(0) */

    case(0):          /* case(3) sets state=0 after processing buffer */
      buf = 0;
      if (CCB->spb_link != 0)          /* if a buffer is ready */
        buf = CCB->spb_link;
      else if (sIfreeQ != 0)           /* if there is a free queue */
        buf = dequeue(sIfreeQ);        /* attempt to get a free buffer */

      if (buf != 0)                    /* if we located a buffer */
      {
      CCB->spb_link = buf;          /* put it on the ccb chain */
      CCB = buf;

        uIFs = 1;                    /* set state to 1 */

        siAh,siAl = CCB->data_buffer;/* must be even */
        sifNb = CCB->max_len - 2;    /* must also be even, and <= 2 */
        enbIdma();                   /* enable the dma microinterrupt */

        sIfreeC -= 1;                /* reduce free buffer count */
        if (sIfreeC < sIfreeL)       /* if it gets too low */
        post(sIfeAh/1);           /* post the free_event */
      }

      if (sItime != 0)           /* if timer for post(data) is on */
        goto case1;              /* Maybe all the buffers are full. */
      break;
```

```
    case(1):    /* case(0) sets state=1, before starting dma. */
      /* we have a timer set which will eventually
         cause us to post(data) */
      if (sItime == 0)        /* timer isn't set, goto state 2 */
      {
        uIFs = 2;
        break;
        }
 case1:                       /* state one enters here */
      if (time_of_day >= sItime) /*  if time is up */
        {
        post(sIdeAh/1);          /* post a data_event */
        sItime = 0;
      }
      break;

    case(2):     /* case 1 sets state=2 when sItime == 0 */
      /* nothing can happen until we get more data */
      break;
    }
}
```

DISTRIBUTION OF THIS REPORT

Defense Advanced Research Projects Agency
Dr. Robert E. Kahn (2)
Dr. Vinton Cerf (1)

Defense Supply Service -- Washington
Jane D. Hensley (1)

Defense Documentation Center
(12)

USC/ISI
Dr. Danny Cohen (2)

MIT/Lincoln Labs
Dr. Clifford J. Weinstein (3)

SRI International
Earl Craighill (1)

Rome Air Development Center
Neil Marples - RBES (1)
Julian Gitlin - DCLD (1)

Defense Communications Agency
Gino Coviello (1)

Bolt Beranek and Newman Inc.
Library
Library, Canoga Park Office
R. Bressler
R. Brooks
P. Carvey
G. Falk
J. Goodhue
E. Hahn
E. Harriman
F. Heart
M. Hoffman
M. Kraley
A. Lake
W. Mann
R. Rettberg
P. Santos
E. Starr
E. Wolf